Queensland University of Technology

Faculty of Science and Engineering

IFN505 Analysis of Programs

Assignment 2: Project

Student Name: MingHuai Tsai (Frank)

Student Number: N10367071

Lecturer Name: Dimitri Perrin

Tutor Name: Jake Bradford

Unit Name: Analysis of Programs

1. Cocktail Shaker Sort

Algorithm1 *CocktailShakerSort*(A[0..n – 1])
// Input: Array $A[0..n-1]$ of numbers
// Output: Array $A[0..n-1]$ of numbers sorted in non-decreasing order
do
  swapped ← **false**
  for $i \leftarrow 0$ **to** $n-2$ **do**
    if $A[i] > A[i+1]$ **then**
      swap ($A[i]$, $A[i+1]$)
      swapped ← **true**
  if not swapped **then**
    return A
  swapped ← **false**
  for $i \leftarrow n-2$ down **to** 0 **do**
    if $A[i] > A[i+1]$ **then**
      swap ($A[i]$, $A[i+1]$)
      swapped ← **true**
while swapped
return A

1.1. Description

This algorithm can sort an array in non-decreasing order. The input is an array which is sorted or not sorted. The output will be a sorted array in non-decreasing order. The array is regarded as the composition of three arrays which are the right sorted array, the middle unsorted array and the left sorted array. All the elements in the right sorted array is greater than the elements in the other two arrays and the elements in the left sorted array is smaller than the elements in the other two arrays. There are two for loops in this algorithm. The first for loop will move the largest element from the middle unsorted array to the right unsorted array. The second loop will move the smallest element from the middle unsorted array to the left unsorted array. The terminal of algorithm is based on “swapped”, when the “swapped” is false, the algorithm will terminate. The occasion that the “swapped” become true is the algorithm executes the swap function.

1.2. Correctness

1.2.1 Invariant

The invariant is “swapped is equal to true”

(1). In the beginning of wile loop, the swapped is true, so the invariant is true.

(2). Although the invariant is violated when entering the loop via “swapped = false”. However, the loop will re-establish the invariant after executing the swap function. As a result, the invariant is true in each iteration.

(3). When the algorithm terminates, assuming that “swapped” is false. The only situation that swapped is false is that the algorithm does not execute the swap function. This means the array is sorted.

1.2.2 Variant

This algorithm can be divided into two situations to prove the total correctness. The first part is the inner for loop, the other part is the outer while loop.

1.2.2.1 Inner loop

The loop variant for first for loop is “n-2-i”.

The loop variant for second for loop is “i”.

1.2.2.2 Outer loop

The variant is the length of the unsorted array. Each iteration, there are two elements will leave the unsorted array. One is swapped into right sorted array, the other one is swapped into left sorted array. Moreover, the length of the array will never be negative, so the length of unsorted array must be zero in the end of algorithm.

1.3. Efficiency

1.3.1 Basic Operation

In this algorithm, the basic operations are comparison $A[i] > A[i+1]$ and swap function $swap(A[i], A[i+1])$. The reason is that they are the all essential operations when sorting an array. In the process of sorting, the program needs to compare the value of two elements and may change the position of them. The action of change the position is executed via “swap” function. The efficiencies of best case, worst case and average case are different, since the program can be stopped immediately when the array is sorted. In some situations, the swap function will not be executed, so the efficiency will also be influenced.

1.3.2 Size of input

The size of input in this algorithm is same as the size of input array.

size = $[(n-1)-0]+1 = n$

1.3.3 Best case

In this algorithm, the array which is increasing order is the best case. The reason is that the swap function does not need to be executed since the element which index is "*n*" always is equal or smaller than the element which index is "*n*+1". Moreover, the algorithm just executes the first for loop once and does not execute the second for loop since the "swapped" is false.

1.3.3.1 calculate the efficiency

Let C_{best}(n) be the numbers of times of comparison and swap performed in the best-case situation.

$C_{best}(1) = 0$

$C_{best}(2) = 1$ (comparison)

$C_{best}(3) = 1 + 1$ (comparison)

$C_{best}(4) = 1 + 1 + 1$ (comparison)

⋮

$$C_{best}(n) = \sum_{i=2}^{n} 1$$

$= n - 2 + 1 = n - 1$ (comparison)

Therefore, $C_{best}(n) = n - 1$

1.3.3.2 Efficiency class

$\forall n, n \geq 0 \Rightarrow \frac{1}{2}n \leq n - 1 \leq 2n$

$\therefore C_{best}(n) \in \Theta(n)$

1.3.4 Worst Case

In this algorithm, the decreasing order array is the worst case. Due to the comparison is always executed "n-1" times, the worst case of algorithm is based on the numbers of time the swap function executed. For the first for loop, in the decreasing order array, the leftmost element in the unsorted array needs to change to the leftmost position of the right sorted array, so the number of times of swap function performed is the highest. Moreover, for the second for loop, in the decreasing order array, the rightmost element in the unsored array needs to change to the rightmost position of the left sorted array, so the number of times of swap function performed is the highest as well. Therefore, the array of the worst case is the decreasing order array.

1.3.4.1 Calculate the efficiency

Let C_{worst}(n) be the numbers of time of comparison and swap function performed in the worst-case situation.

$C_{worst}(1) = 0$

$C_{worst}(2) = 1 + 1(once\ comparion\ and\ swap) + 1(check) = 3$

$C_{worst}(3) = 2 + 2(first\ loop\ needs\ two\ comparison\ and\ two\ swap)$
$+ 2 + 1(second\ loop\ needs\ two\ comparison\ and\ one\ swap)$
$+ 2\ (comparison\ for\ checking) = 9$

$C_{worst}(4) = (3 + 3) + (3 + 2) + (3 + 1) + 3 = 18$

…

$C_{worst}(n) = (n - 1 + n - 1) + (n - 1 + n - 2) + \cdots + (n - 1 + 1) + (n - 1)$

$= (n - 1) * (n - 1) + (n - 1 + n - 2 + n - 3 + \cdots + 1) + (n - 1)$

$= (n - 1) * (n - 1) + \left(\frac{n(n-1)}{2}\right) + (n - 1)$

$= (n - 1) * \left[(n - 1) + \frac{n}{2} + 1\right]$

$= \frac{3n(n-1)}{2}$

1.3.4.2 Efficiency class

$\forall n, n \geq 4 \Rightarrow n^2 \leq \frac{3(n^2 - n)}{2} \leq 2n^2$

$\therefore C_{worst}(n)\ \in \Theta(n^2)$

1.3.5 Average case

1.3.5.1 Calculate the efficiency

As mentioned before, the maximum number of times of executing swap function is n, and it will decrease after executing once. Moreover, the number of times of comparison performed always is n-1. Therefore, the situation can be discussed as “maximum swap times = n-1”, “maximum swap times = n-2”, …, “maximum swap times = 1” and “maximum swap times = 0”.

(1). Maximum swap times = n-1

This is the worst case

Total:

$$(\text{n-1})*(\text{n-1}) + \sum_{\text{i=1}}^{\text{n-1}} \text{i} + (\text{n-1})$$

(2). Maximum swap times = n-2

Comparison: (n-2) * (n-1)

Swap: (n-2) + (n-3) + … + 2 + 1

Check: n-1

Total:

$$= (\text{n-2})*(\text{n-1}) + \sum_{\text{i=1}}^{\text{n-2}} \text{i} + (\text{n-1})$$

(3). Maximum swap times = n-3

Comparison: (n-3) * (n-1)

Swap: (n-3) + (n-4) + … + 2 + 1

Check: n-1

Total:

$$= (\text{n-3})*(\text{n-1}) + \sum_{\text{i=1}}^{\text{n-3}} \text{i} + (\text{n-1})$$

…

(n-2). Maximum swap times = 2

Comparison: 2 * (n-1)

Swap: (2 + 1)

Check: n-1

Total

$$= (2)*(\text{n-1}) + \sum_{\text{i=1}}^{2} \text{i} + (\text{n-1})$$

(n-1). Maximum swap times = 1

Comparison: 1* (n-1)

Swap: 1

Check: n-1

Total

$$= (1)*(\text{n-1}) + \sum_{\text{i=1}}^{1} \text{i} + (\text{n-1})$$

(n). Maximum swap times = 0

This is best case.

Comparison: 0* (n-1)

Swap: 0

Check: n-1

Total

$= (n\text{-}1)$

Summary:

Total of (1) + (2) + … + (n-2) + (n-1) + (n)

$$= (n\text{-}1)\sum_{i=1}^{n-1} i + \sum_{j=1}^{n-1}\sum_{i=1}^{j} i + (n\text{-}1)\sum_{i=1}^{n-1} 1 + (n\text{-}1)$$

$$= (n\text{-}1)(\frac{n(n\text{-}1)}{2}) + \sum_{j=1}^{n-1}\frac{j(j+1)}{2} + (n\text{-}1)(n\text{-}1) + (n\text{-}1)$$

$$= (n\text{-}1)(\frac{n(n\text{-}1)}{2}) + \frac{1}{2}\sum_{j=1}^{n-1} j^2 + \frac{1}{2}\sum_{j=1}^{n-1} j + (n\text{-}1)(n\text{-}1) + (n\text{-}1)$$

$$\approx (n\text{-}1)(\frac{n(n\text{-}1)}{2}) + \frac{(n\text{-}1)^3}{6} + \frac{(n\text{-}1)^2}{6} + (n\text{-}1)(n\text{-}1) + (n\text{-}1)$$

$$= \frac{n^3\text{-}2n^2\text{-}n}{2} + \frac{n^3\text{-}2n^2 + n + 2}{6} + n^2\text{-}2n + 1 + n\text{-}1$$

$$\approx \frac{4n^3}{6}$$

The number of times of comparison and swap function performed is $\frac{4n^3}{6}$.

The total cases is n.

The average number of times of basic operation performed is

$$= \frac{total\ numbers\ of\ times\ of\ comparison\ +\ total\ numbers\ of\ times\ of\ swap}{total\ cases}$$

$$= \frac{\frac{4n^3}{6}}{n}$$

$$\approx n^2$$

1.3.5.2 Efficiency class

$\forall n, n \geq 1 \Rightarrow \frac{n^2}{2} \leq n^2 \leq 2n^2$

$\therefore C_{average}(n) \in \Theta(n^2)$

2. Insertion Sort

Algorithm2 *InsertionSort* ($A[0..n-1]$)
// Input: Array $A[0..n-1]$ of numbers
// Output: Array $A[0..n-1]$ of numbers sorted in non-decreasing order

$i \leftarrow 1$
while $i < n$ **do**
  $j \leftarrow i$
  while $j > 0$ **and** $A[j-1] > A[j]$ **do**
    swap ($A[j]$, $A[j-1]$)
    $j \leftarrow j-1$
  $i \leftarrow i+1$
return A

2.1 Description

This algorithm can sort an array in non-decreasing order. The input is an array which is sorted or not sorted. The output will be a sorted array in non-decreasing order. The array can regard as two parts including the left part and the right part. The subarray in the left part is sorted and the subarray in the right part is unsorted. The inner while loop will move the leftmost element in the right unsorted array to the appropriate position in the left array in each iteration. The array will terminate when the unsorted array is empty.

2.2. Correctness

2.2.1 Invariant

The loop invariant is

$(\forall x: 0,1,2,\dots i-1, A[x-1] \leq A[x]) \wedge ([A[l]|0 \leq l \leq n-1] = [\mathrm{A}_0[m]|0 \leq m \leq n-1])$

The A_0 represent the original array A.

(1). In the beginning of loop, i is 1. The array A only have one element, so the invariant is true.

(2). The loop preserves the invariant by swapping A[i] to appropriate position.

Let k be the appropriate position. There are three situations for k.

(2.1) $k = 0$

$(\forall x: 1,2,\dots i, A[x-1] \leq A[x]) \wedge (A[0] \leq A[1])$

$\equiv (\forall x: 0,1,2,\dots i, A[x-1] \leq A[x])$

$\equiv (\forall x: 0,1,2,\dots (i+1)-1, A[x-1] \leq A[x])$

$(\forall x: 0,1,2,\dots i-1, A[x-1] \leq A[x])$ assign $i = i+1$

(2.2) $0 < k < i$

$(\forall x: 0,1,2, \dots \mathrm{k}-1, A[x-1] \leq A[x]) \wedge (\forall x: \mathrm{k}+1, \dots i, A[x-1] \leq A[x]) \wedge (\mathrm{A[k-1]} < \mathrm{A[k]} < \mathrm{A[k+1]})$

$\equiv (\forall x: 0,1,2, \dots i, A[x-1] \leq A[x])$

$\equiv (\forall x: 0,1,2, \dots (i+1)-1, A[x-1] \leq A[x])$

$(\forall x: 0,1,2, \dots i-1, A[x-1] \leq A[x])$ assign $i = i+1$

(2.3) $k = i$

$(\forall x: 0,1,2, \dots i-1, A[x-1] \leq A[x]) \wedge (A[i-1] \leq A[i])$

$\equiv (\forall x: 0,1,2, \dots i, A[x-1] \leq A[x])$

$\equiv (\forall x: 0,1,2, \dots (i+1)-1, A[x-1] \leq A[x])$

$(\forall x: 0,1,2, \dots i-1, A[x-1] \leq A[x])$ assign $i = i+1$

(3). when the loop terminates, assuming i is equal to n

$(\forall x: 0,1,2, \dots n-1, A[x-1] \leq A[x])$ assign $\mathrm{n} = \mathrm{i}$

2.2.2 Variant

This algorithm can be divided into two situations to prove the total correctness. The first part is the inner loop, the other part is the outer loop.

2.2.2.1 Inner loop

The loop variant of inner loop is j.

1. In the beginning, j is equal to i, this means the j is non-negative number.

2. In each iteration, because “j = j - 1”, j is decreased.

3. Therefore, the inside loop can always terminate.

2.2.2.2 Outer loop

The loop variant is the length of unsorted subarray in array.

Proof:

1. The length of unsorted subarray in array is same as the length of array in the beginning

2. In this algorithm, for each iteration, there is an element in the unsorted subarray will be swapped into sorted subarray. Therefore, the length will be divided by 1 in each iteration. Moreover, the length will not be negative, so it will become zero in the end.

3. As a result, the algorithm can always terminate.

2.3. Efficiency

2.3.1 Basic Operation

In this algorithm, the basic operations are comparison $A[j-1] > A[j]$and swap function $swap(A[j], A[j-1])$. The reason is that they are the all essential operations when sorting an array.

In the process of sorting, the program needs to compare the value of two elements and may change the position of them which is “swap”.

2.3.2 Input size

The size of input in this algorithm is same as the size of input array.

size = $[(n - 1) - 0] + 1 = n$

2.3.3 Best case

In this algorithm, the best case is the array with non-decreasing order, since every element just needs to do comparison $A[j-1] > A[j]$ once. Moreover, they do not need to do swap function, because $A[j-1]$ always equal or smaller than $A[j]$.

2.3.3.1 Calculate the efficiency

Let $I_{best}(n)$ be the numbers of times of comparison and swap performed in the best-case situation.

$I_{best}(1) = 0$ (Do not need to compare and swap)

$I_{best}(2) = 1(comparsion)$

$I_{best}(3) = 1 + 1(comparison)$

$I_{best}(4) = 1 + 1 + 1(comparison)$

…

$I_{best}(n) = 1 + 1 + 1 \ldots + 1(comparison)$

$$= \sum_{i=2}^{n} 1$$

$= n - 2 + 1 = n - 1$ (comparison)

Therefore, $I_{best}(n) = n - 1$

2.3.3.2 Efficiency class

$\forall n, n \geq 0 \Rightarrow \frac{1}{2}n \leq n - 1 \leq 2n$

$\therefore I_{best}(n) \in \Theta(n)$

2.3.4 Worst Case

In this algorithm, the decreasing array is the worst case. The reason is that every element in the right unsorted array needs to move from the rightmost position of the left sorted array to the leftmost position. As a result, it needs do the most times swap and comparison

.

2.3.4.1 Calculate the efficiency

Let $I_{worst}(1)$ be the numbers of time of comparison and swap function performed in the worst-case situation.

$I_{worst}(1) = 0$

$I_{worst}(2) = 1(comparsion) + 1(swap) + I_{worst}(1) = 2 + 0 = 2$

$I_{worst}(3) = 4\ (2\ for\ comparison, 2\ for\ swap) + I_{worst}(2) = 4 + 2 = 6$

$I_{worst}(4) = 6\ (3\ for\ comparison, 3\ for\ swap) + I_{worst}(3) = 6 + 6 = 12$

…

$I_{worst}(n) = 2 * (n-1)(n-1\ for\ comparison, n-1\ for\ swap\ ) + I_{worst}(n-1)$

$= 2 * (n-1) + 2 * (n-2) + \cdots + 2 * 2 + 2 * 1$

$= 2 * [(n-1) + (n-2) + \cdots + 2 + 1]$

$= 2 * \sum_{i=1}^{n-1} i$

$= 2 * \frac{(n-1+1)(n-1)}{2}$

$= n * (n-1)$

$= n^2 - n$

2.3.4.2 Efficiency class

$\forall n, n \geq 2 \Rightarrow \frac{n^2}{2} \leq n^2 - n \leq n^2$

$\therefore I_{worst}(n)\ \in \Theta(n^2)$

2.3.5 Average case

Firstly, focusing on the random element $A[i]$ in the array. Then, comparing with the sorted array in front, the value of this element can be "the largest", "second largest", "third largest" …, "ith largest = second smallest" and "(i+1)th largest = smallest". Therefore, it totally has $i + 1$ cases.
Secondly, calculating the numbers of times of comparison performed in every case and then summary the numbers.

Value of element A[i]	Numbers of times
Largest	1 (compare with A[i-1])
Second largest	2 (compare with A[i-1], A[i-2])
Third largest	3 (compare with A[i-1], A[i-2], A[i-3])
…	
Third smallest = (i-1)th largest	i-1 (compare with A[i-1], A[i-2], …, A[i-(i-1)])

Second smallest = i^{th} largest	i (compare with A[i-1], A[i-2], …, A[1], A[0])
Smallest = $(i+1)^{th}$ largest	i (compare with A[i-1], A[i-2], …, A[1], A[0])

The numbers in "Second smallest" case and "Smallest" case are the same is because they all do the comparison with A[0] in these two situations. The difference is that A[i] need to be changed the position with A[0] or not, and this will be discussed later in calculating the numbers of times of swap function performed.

The total numbers of A[i]

$$= 1 + 2 + 3 + \cdots + i - 1 + i + i$$

$$= \frac{i * (1 + i)}{2} + i$$

Thirdly, calculating the numbers of times of swap function executed in every case and then summary numbers.

Value of element A[i]	Numbers of times
Largest	0 (Do not do the swap)
Second largest	1 (swap with A[i-1])
Third largest	2 (swap with A[i-1], A[i-2])
…	…
Third smallest = $(i-1)^{th}$ largest	i-2 (swap with A[i-1], A[i-2], …, A[2])
Second smallest = i^{th} largest	i-1 (swap with A[i-1], A[i-2], …, A[2], A[1])
Smallest = "$(i+1)^{th}$ largest	i (swap with A[i-1], A[i-2], …, A[2], A[1], A[0])

The total of numbers

$$= 0 + 1 + 2 + \cdots + (i - 1) + i$$

$$= \frac{(i + 0) * (i - 0 + 1)}{2}$$

$$= \frac{i * (i + 1)}{2}$$

Fourthly, add the numbers of times of comparison performed and the numbers of times of swap function performed. The result will be the total numbers of times of basic operations performed in the position "i".

$$= \frac{i * (1 + i)}{2} + i + \frac{i * (i + 1)}{2}$$

$$= i * (i + 1) + i$$

$$= i^2 + 2i$$

Fifthly, the result in the fourth step divide total case. The new result will be the average numbers of times of basic operations performed in the position "i".

$$= \frac{total\ numbers\ of\ times\ of\ comparison\ +\ total\ numbers\ of\ times\ of\ swap}{total\ cases}$$

$$= \frac{i^2 + 2i}{i + 1}$$

$$= \frac{(i+1)(i+1) - 1}{i + 1}$$

$$= i + 1 - \frac{1}{i+1}$$

Finally,

let $I_{average}(n)$ be the numbers of times of basic operation performed in the average case

$I_{average}(n)$

$$= \sum_{i=0}^{n-1} (i + 1 - \frac{1}{i+1})$$

$$= \sum_{i=0}^{n-1} i + \sum_{i=0}^{n-1} 1 - \sum_{i=0}^{n-1} \frac{1}{i+1}$$

In this situation, the last big sigma will not influence the result since the value of it will become very small when n become larger. Therefore, it is not necessary to be calculated.

$$= \frac{(n-1)(n-1+1)}{2} + (n - 1 - 0 + 1) - \sum_{i=0}^{n-1} \frac{1}{i+1}$$

$$= \frac{(n-1)(n-1+1)}{2} + (n - 1 - 0 + 1) - \sum_{i=0}^{n-1} \frac{1}{i+1}$$

$$= \frac{n^2 - n}{2} + n - \sum_{i=0}^{n-1} \frac{1}{i+1}$$

$$= \frac{n^2 + n}{2} - \sum_{i=0}^{n-1} \frac{1}{i+1}$$

$$\approx \frac{n^2 + 3n}{2}$$

Therefore, the numbers of times of basic operations performed in average case is $\frac{n^2+3n}{2}$

2.3.5.2 Efficiency class

$$\forall n, n \geq 0, \frac{n^2}{2} \leq \frac{n^2 + 3n}{2} \leq 2n^2$$

$$\therefore I_{average}(n)\ \in \Theta(n^2)$$

3. Methodology

3.1 Environment

The program language is C.

3.1.1 Computer environment

- OS: Windows 10 Enterprise 64-bit
- CPU: Intel Core i7-6700
- Memory: 16GB

3.1.2 Compile

- Compiler: gcc version 8.2.0 (MinGW.org GCC-8.2.0-3)

The command below is used to compile the program.

$ gcc main.c -o main.exe

3.2 Implementation of algorithm

The program includes 2 files which are “main.c” and “function.h”. The entry point of program is in “main.c” and it also implies two algorithms.

3.2.1 Data structure and Swap function

To implement the algorithm, data structure “sort_results” is defined. It includes “counts” which is used to count the numbers of time of basic operation performed and “sorted_array” which is used to store the sorted array (figure 3.1).

```
10  typedef struct sort_results {
11      long long count;
12      int *sorted_array;
13  } sort_result;
```

Figure 3.1

The swap function in “function.h” is shown in figure 3.2.

```
30  void swap(int *ary, int index, int index2) {
31      int temp = ary[index];
32      ary[index] = ary[index2];
33      ary[index2] = temp;
34  }
```

Figure 3.2

3.2.2 Cocktail Shaker Sort

Figure 3.3 shows the implement of cocktail shaker sort algorithm. The cs1 will store count and sorted_array. The sorted_array is the main output of the algorithm. The count is the number of times of basic operation performed. There are two basic operation in this algorithm, which are swap function and comparison.

3.2.2.1 Counter

In line 191 and 206, the “count” variable will calculate the number of times of comparison performed. In line 195 and 210, the “count” variable will calculate the number of times of swap function performed.

```
183  sort_result* cocktail_shaker_sort(int *ary, int len) {
184      int swapped = 0;
185      long long count = 0;
186      sort_result *cs1 = malloc(sizeof(sort_result));
187
188      do {
189          swapped = 0;
190          for(int i = 0; i <= len - 2; i++) {
191              count++;
192              if(ary[i] > ary[i+1]) {
193                  swap(ary, i, i + 1);
194                  swapped = 1;
195                  count++;
196              }
197          }
198          if(!swapped){
199              cs1->count = count;
200              cs1->sorted_array = malloc(len * sizeof(int));
201              cs1->sorted_array = ary;
202              return cs1;
203          }
204          swapped = 0;
205          for(int i = len-2; i >= 0; i--){
206              count++;
207              if(ary[i] > ary[i+1]) {
208                  swap(ary, i, i + 1);
209                  swapped = 1;
210                  count++;
211              }
212          }
213      } while (swapped);
214
215      cs1->count = count;
216      cs1->sorted_array = malloc(len * sizeof(int));
217      cs1->sorted_array = ary;
218      return cs1;
219  }
```

Figure 3.3

3.2.3 Insertion sort

Figure 3.4 show the implement of insertion sort. The cs1 will store count and sorted_array. The sorted_array is the main output of the algorithm. The count is the number of times of basic operation performed. There are two basic operation in this algorithm, which are swap function and comparison. In this program, in order to calculate the number, I have changed a part of algorithm. As shown in figure 3.4, in line 228 and 230, I separated the “while j > 0 and A[j – 1] > A[j] do” into “while(i < len)” and “if (ary[j-1], ary[j])”. The program also has the same output as the algorithm, because when the “if” statement is false, it will execute “break” operator. Therefore, the program will exit loop, and the algorithm will also exit the loop when “ary[j-1], ary[j]” is false.

3.2.3.1 Counter

In this program, in line 229, the “count” variable will calculate the number of times of comparison performed and the number of times of swap function performed will be calculated in line 233.

```
221  sort_result* insertion_sort(int *ary, int len) {
222      int i = 0, j;
223      long long count = 0;
224      sort_result *cs1 = malloc(sizeof(sort_result));
225
226      while(i < len){
227          j = i;
228          while(j > 0){
229              count++;
230              if(ary[j-1] > ary[j]) {
231                  swap(ary, j - 1, j);
232                  j = j - 1;
233                  count++;
234              } else {
235                  break;
236              }
237          }
238          i = i + 1;
239      }
240
241      cs1->count = count;
242      cs1->sorted_array = malloc(len * sizeof(int));
243      cs1->sorted_array = ary;
244
245      return cs1;
246  }
```

Figure 3.4

3.2.4 Correctly

The function “verify” in the “function.h” is to ensure that the output of the program is correct (figure 3.5). Function “verify” check “ary[i]” and its next element “ary[i+1]”, because the output of algorithm is non-decreasing order array. When the element is greater than the next element, the function returns 0 which means false since the array is not non-decreasing order. The usage of function will be mentioned later.

```
36   int verify(int *ary, int len) {
37       for(int i = 0; i < len-1; i++) {
38           if(ary[i] > ary[i+1]) {
39               return 0;
40           }
41       }
42       return 1;
43   }
```

Figure 3.5

3.3 Testing data

The functions which can provide the testing data are all in “function.h”.

3.3.1 Average case

In average case, the testing data must be generated randomly, so the function “get_array” is provided to require the array. The function “rand()” can randomly generate the number. The elements in this array are greater than 0 and are smaller than “INT32_MAX”which is equal to 2,147,483,647 since it is the largest number that the program can execute and (figure 3.6).

```
45  int * get_array(int len) {
46      int* array = malloc(len * sizeof(int));
47      for(int i = 0; i < len; i++) {
48          array[i] = rand() % INT32_MAX;
49      }
50      return array;
51  }
```

Figure 3.6

To ensure the array is random, it needs to set seed. In C, the function “void srand(unsigned int seed);” is adopted to set seed. As shown in figure 3.7, in “main.c”, the seed in this program is “time(0)” in line 64, because the time is always changed.

```
63      // ensure the array is random
64      srand(time(0));
65
```

Figure 3.7

In order to execute “rand” and “srand” the “stdlib.h” needs to be imported.

3.3.2 Worst case

In worst case, as we have discussed before, the array is decreasing order. The program provides “get_worst_case_array” to supply the array. The element in index “0” is the largest, and the element in index “len-1” is the smallest (figure 3.8).

```
53  // Worst case means the element in index "0" is the biggest, and in index "len-1" is the smallest
54  int * get_worst_case_array(int len) {
55      int *array = malloc(len * sizeof(int));
56      for(int i = 0; i < len; i++){
57          array[i] = len - i;
58      }
59      return array;
60  }
```

Figure 3.8

3.3.3 Best case

In best case, as we have discussed before, the array is increasing order. The program provides “get_best_case_array” to supply the array. The element in index “0” is the smallest, and the element in index “len-1” is the largest (figure 3.9).

```
62  // Worst case means the element in index "0" is the smallest, and in index "len-1" is the biggest
63  int * get_best_case_array(int len) {
64      int *array = malloc(len * sizeof(int));
65      for(int i = 0; i < len; i++) {
66          array[i] = i+1;
67      }
68      return array;
69  }
```

Figure 3.9

3.3.4 Size of Testing Array

The size of testing array is decided by two variables, which are LOOP_TIMES and SIZE_TIMES. The size is start from 0 * SIZE_TIMES to LOOP_TIMES * SIZE_TIMES, so it increases SIZE_TIMES for each loop.

As shown in figure 3.10, the LOOP_TIMES and SIZE_TIMES are defined in the beginning of the program as the global variable in the “main.c”. Then, they are adopted in the measurement function and comparison function (figure 3.11 and 3.12).

```
6   //set loopstime and size_times
7   // example => loop_times = 10, size_times = 50
8   // do loop for 11 times, array size from 0 to 500 => 0, 50, 100, 150 ..., 450, 500
9   #define LOOP_TIMES 50
10  #define SIZE_TIMES 1000
```

Figure 3.10

```
267     for(int i = 0; i <= LOOP_TIMES; i++) {
268         sum_total_t = 0;
269         sum_count = 0;
270         for (int j = 0; j < AVERAGE_TIMES; j++) {
271             // Get size
272             len = i * SIZE_TIMES;
```

Figure 3.11

```
331     for(int i = 0; i <= LOOP_TIMES; i++) {
332         sum_total_t = 0;
333         sum_total_t_b = 0;
334         sum_count = 0;
335         sum_count_b = 0;
336         for (int j = 0; j < AVERAGE_TIMES; j++) {
337             // Get size
338             len = i * SIZE_TIMES;
```

Figure 3.12

3.4 Time measurement

The “time.h” is imported in line 3 to measure the execution time of the algorithms (figure 3.13.)

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <time.h>
4   #include "function.h"
```

Figure 3.13

In the function of measurement, the “clock_gettime” is called to measure the time. The first one represents the start time (line 278) and the second one represents the end time (line 280). After the execution, the total time is end time minus start time and is stored in total_t (line 282 and 283) (figure 3.14).

```
277             // Start
278             if (clock_gettime(CLOCK_MONOTONIC, &start_t) == -1) exit(0);
279             sort_results = funcs.sort_functions[funcs.sort_approaches[0]](a, len);
280             if (clock_gettime(CLOCK_MONOTONIC, &end_t) == -1) exit(0);
281
282             total_t = ((double) end_t.tv_sec + 1.0e-9 * (double) end_t.tv_nsec) -
283                       ((double) start_t.tv_sec + 1.0e-9 * (double) start_t.tv_nsec);
```

Figure 3.14

In the function of comparison, this activity will be performed twice to calculate the execution time of different algorithms (figure 3.15).

```
344             // Start array_a
345             if (clock_gettime(CLOCK_MONOTONIC, &start_t) == -1) exit(0);
346             s_results = funcs.sort_functions[funcs.sort_approaches[0]](array_a, len);
347             if (clock_gettime(CLOCK_MONOTONIC, &end_t) == -1) exit(0);
348             total_t = ((double) end_t.tv_sec + 1.0e-9 * (double) end_t.tv_nsec) -
349                       ((double) start_t.tv_sec + 1.0e-9 * (double) start_t.tv_nsec);
350             //printf("A Index = %d is done Time = %.7f Count = %lli \n", len, total_t, s_re
351             sum_total_t += total_t;
352             sum_count += s_results->count;
353
354             // start array_b
355             if (clock_gettime(CLOCK_MONOTONIC, &start_t_b) == -1) exit(0);
356             s_results_b = funcs.sort_functions[funcs.sort_approaches[1]](array_b, len);
357             if (clock_gettime(CLOCK_MONOTONIC, &end_t_b) == -1) exit(0);
358             total_t_b = ((double) end_t_b.tv_sec + 1.0e-9 * (double) end_t_b.tv_nsec) -
359                         ((double) start_t_b.tv_sec + 1.0e-9 * (double) start_t_b.tv_nsec);
360             //printf("B Index = %d is done Time = %.7f Count = %lli \n", len, total_t_b, s_
361             //printf("\n");
362             sum_total_t_b += total_t_b;
363             sum_count_b += s_results_b->count;
```

Figure 3.15

3.5 Measurement and Comparison

The program provides two function in the “main.c” to measure the algorithms. The first function is “measure_function”, it measures the time and basic operation of an algorithm. The second function is “compare_function”, it provides the comparison of the results of two algorithms.

3.5.1 Data structure

In order to pass the function pointer into the measurement function and comparison function, the “function_structre” is declared in this program (figure 3.16). The meanings of each variable are shown in table 1

```
22  // get_array_approaches: 0 -> random array, 1 -> worst case array, 2 -> best case array
23  // sort_approaches: 0 -> cocktail_shaker_sort, 1 -> insertion_sort
24  typedef struct function_structs{
25      sort_function *sort_functions;
26      get_array_function *get_array_functions;
27      int get_array_approaches;
28      int *sort_approaches;
29  } function_struct;
```

Figure 3.16

Variable	Meaning
sort_functions	Store the functions which imply the alogorithm.
get_array_functions	Store the function which can generate the testing array.
get_array_approaches	Integer type, it is used to choose the get_array function.
sort_approaches	Integer type, it is used to choose the algorithm

Table 1

In figure 3.17, the algorithms and functions for generating the testing array are store in the “funcs” and the relationship between functions and variable is shown in table 2 and 3.

```
49      sort_function sort_functions[] = {cocktail_shaker_sort, insertion_sort};
50      get_array_function get_array_functions[] = {get_array, get_worst_case_array, get_best_case_array};
51
52      // set function pointer
53      function_struct *funcs = malloc(sizeof(function_struct));
54      funcs->sort_functions = malloc(2 * sizeof(sort_function));
55      for(int i = 0; i < 2; i++) {
56          funcs->sort_functions[i] = sort_functions[i];
57      }
58      funcs->get_array_functions = malloc(3 * sizeof(get_array_function));
59      for(int i = 0;i < 3; i++) {
60          funcs->get_array_functions[i] = get_array_functions[i];
61      }
```

Figure 3.17

funcs.sort_functions	Functions
funcs.sort_functions[0]	Cocktail Shaker Sort
funcs.sort_functions[1]	Insertion Sort

Table 2

funcs.get_array_function	Functions
funcs.get_array_function[0]	get_array
funcs.get_array_function[1]	get_worst_case_array
funcs.get_array_function[2]	get_best_case_array

Table 3

3.5.2 Measurement function

The function which is adopted to measure the time and basic operation is “measure_function”.

```
267     for(int i = 0; i <= LOOP_TIMES; i++) {
268         sum_total_t = 0;
269         sum_count = 0;
270         for (int j = 0; j < AVERAGE_TIMES; j++) {
271             // Get size
272             len = i * SIZE_TIMES;
273
274             // Get array
275             a = funcs.get_array_functions[funcs.get_array_approaches](len);
276
277             // Start
278             if (clock_gettime(CLOCK_MONOTONIC, &start_t) == -1) exit(0);
279             sort_results = funcs.sort_functions[funcs.sort_approaches[0]](a, len);
280             if (clock_gettime(CLOCK_MONOTONIC, &end_t) == -1) exit(0);
281
282             total_t = ((double) end_t.tv_sec + 1.0e-9 * (double) end_t.tv_nsec) -
283                       ((double) start_t.tv_sec + 1.0e-9 * (double) start_t.tv_nsec);
284             //printf("Index = %d is done Time = %.7f Count = %lli \n", len, total_t,
285             sum_total_t += total_t;
286             sum_count += sort_results->count;
287
288             // verify array is sorted
289             is_sorted = verify(sort_results->sorted_array, len);
290             if (!is_sorted) {
291                 printf("Index = %d Is not sorted\n", len);
292             }
293
294             free(a);
295         }
296         // do average
297         sum_total_t /= AVERAGE_TIMES;
298         sum_count /= AVERAGE_TIMES;
299
300         // Printf
301         printf("%5d \t %.7f \t %10lli\n", len, sum_total_t, sum_count);
302
303         // output file
304         fprintf(fptr, "%d %.7f %lli\n", len, sum_total_t, sum_count);
305     }
```

Figure 3.18

3.5.2.1 Average

To improve the correctness, the function which has same size of input will do AVERAGE_TIMES times. Then, the program will do the summary of the result of each execution and do average and store in “sum_total_t” and “sum_count”. The equation is shown below.

$$Average\ time/basic\ operation = \frac{summary\ of\ time/basic\ operation}{AVERAGE\_TIMES}$$

In figure 3.18, the program does the summary of the result of each execution in line 285 and 286 and the average is calculated in line 297 and 298.

3.5.2.2 Timer

The time measurement approach is mentioned in the section 3.4. In figure 3.18, the timer is started in line 278, then the function of algorithm will be started as well. After the function finish, the timer will be stopped in line 280. Therefore, the measurement of time is accurate since the timer is started before the algorithms execute and is stopped after the algorithm finish immediately.

3.5.2.3 Get testing array

As shown in figure 3.18, the get_array function is called in line 275 and the testing array will be stored in “a”. The relationship between “a” and “funcs.get_array_approaches” is shown in table 4.

funcs.get_array_approaches	a
0	Array for measuring average case
1	Array for measuring worst case
2	Array for measuring best case

Table 4

3.5.2.4 Perform the algorithm

The functions which are the implementation of algorithms will be called in line 279. The result of the function will be stored in sort_results. The relationship between “Action” and “funcs.sort_approaches[0]” is shown in table 5.

funcs.sort_approaches[0]	Action
0	Do cocktail shaker sort
1	Do insertion sort

Table 5

3.5.2.5 Verify

The verify function which has been described in section 3.2.4 will be called in line 289 to check the result (figure 3.18). If the array is not sorted, the program will print which length of array is not sorted.

3.5.3 Comparison function

The function which is adopted to compare the time and numbers of times of basic operation performed is “comparison_function”.

The comparison function performs two different algorithms with same input to compare the time and the number of times of the basic operation performed. The content of this function is similar as measurement function and the action of doing average, timer and check are similar as well so they will not be described again.

```
340             // Get array
341             array_a = funcs.get_array_functions[funcs.get_array_approaches](len);
342             array_b = copy_array(array_a, len);
343
344             // Start array_a
345             if (clock_gettime(CLOCK_MONOTONIC, &start_t) == -1) exit(0);
346             s_results = funcs.sort_functions[funcs.sort_approaches[0]](array_a, len);
347             if (clock_gettime(CLOCK_MONOTONIC, &end_t) == -1) exit(0);
348             total_t = ((double) end_t.tv_sec + 1.0e-9 * (double) end_t.tv_nsec) -
349                       ((double) start_t.tv_sec + 1.0e-9 * (double) start_t.tv_nsec);
350             //printf("A Index = %d is done Time = %.7f Count = %lli \n", len, total_t, s_resu
351             sum_total_t += total_t;
352             sum_count += s_results->count;
353
354             // start array_b
355             if (clock_gettime(CLOCK_MONOTONIC, &start_t_b) == -1) exit(0);
356             s_results_b = funcs.sort_functions[funcs.sort_approaches[1]](array_b, len);
357             if (clock_gettime(CLOCK_MONOTONIC, &end_t_b) == -1) exit(0);
358             total_t_b = ((double) end_t_b.tv_sec + 1.0e-9 * (double) end_t_b.tv_nsec) -
359                         ((double) start_t_b.tv_sec + 1.0e-9 * (double) start_t_b.tv_nsec);
360             //printf("B Index = %d is done Time = %.7f Count = %lli \n", len, total_t_b, s_re
361             //printf("\n");
362             sum_total_t_b += total_t_b;
363             sum_count_b += s_results_b->count;
```

Figure 3.19

3.5.3.1 Get testing data

As shown in figure 3.19, the get_array function is called in line 341 and the testing array will be stored in “a”. The relationship between “array_a” and “funcs.get_array_approaches” is shown in table 6.

funcs.get_array_approaches	array_a
0	Array for measuring average case
1	Array for measuring worst case
2	Array for measuring best case

Table 6

3.5.3.2 Same testing data

The deep copy method is adopted since the arrays need to be same and cannot be influenced by another algorithm. The copied array will be stored in array_b and wiil be the input of algorithm2. The origin array will be stored in array_a and will be the input of algorithm1.

Figure 3.20 show the copy function and figure 3.21 shows the implementation in line 342.

```
71  // Deep copy
72  int * copy_array(int *origin, int len) {
73      int *b = malloc(len * sizeof(int));
74      memcpy(b, origin, len * sizeof(int));
75      return b;
76  }
```

Figure 3.20

```
340         // Get array
341         array_a = funcs.get_array_functions[funcs.get_array_approaches](len);
342         array_b = copy_array(array_a, len);
```

Figure 3.21

3.5.3.3 Perform the algorithm

In comparison function, both algorithms need to be called, so the “funcs.sort_approaches” variable will store different number to call the algorithms. As shown in figure 3.18, line 346 and line 356, the program calls two different functions and passes different array. The relationship between “Action” and “funcs.sort_approaches” is shown in table 7 and 8.

funcs.sort_approaches[0]	Action
0	Do cocktail shaker sort
1	Do insertion sort

Table 7

funcs.sort_approaches[1]	Action
0	Do cocktail shaker sort
1	Do insertion sort

Table 8

3.5.4 Guide for Using function

3.5.4.1 Measure Function

The figure 3.22 shows the example of measuring the best case of cocktail shaker sort. Following table 2 and table 3. The index of cocktail shaker sort is 0, so 0 is assigned to “funcs->sort_approaches[0]”. The index of get best case array is 2, so 2 is assigned to “funcs->get_array_approaches”.

```
96      // The best case
97      printf("====================================\n");
98      printf("=======The Best Case  Start========\n");
99      printf("====================================\n");
100
101     funcs->sort_approaches = (int *)malloc(1 * sizeof(int));
102     funcs->sort_approaches[0] = 0;
103     funcs->get_array_approaches = 2;
104     measure_function(*funcs, "best_result_cocktail.txt");
```

Figure 3.22

3.5.4.1 Comparison Function

The figure 3.23 shows the example of measuring the comparison of average case. Following table2 and table 3. The index of cocktail shaker sort is 0, so 0 is assigned to “funcs->sort_approaches[0]”. The index of insertion sort is 1, so 1 is assigned to “funcs->sort_approaches[1]”. The index of get best case array is 0, so 0 is assigned to “funcs->get_array_approaches”.

```
146     //Comparison
147     printf("====================================\n");
148     printf("=============Average===============\n");
149     printf("====================================\n");
150
151     funcs->sort_approaches = (int *)malloc(2 * sizeof(int));
152     funcs->sort_approaches[0] = 0;
153     funcs->sort_approaches[1] = 1;
154     funcs->get_array_approaches = 0;
155     compare_function(*funcs, "avg_result_comparison.txt");
```

Figure 3.23

3.5.5 Output file

The program provides the *.txt file for analysing. The relationship between case and file name is shown in table 9.

Case	File name
Average case of Cocktail Shaker Sort	avg_result_cocktail.txt
Worst case of Cocktail Shaker Sort	worst_result_cocktail.txt
Best case of Cocktail Shaker Sort	best_result_cocktail.txt
Average case of Insertion Sort	avg_result_insertion.txt
Worst case of Insertion Sort	worst_result_insertion.txt
Best case of Insertion Sort	best_result_insertion.txt
Average case of comparison	avg_result_comparison.txt
Worst case of comparison	worst_result_comparison.txt
Best case of comparison	best_result_comparison.txt

Table 9

4.Result

There are nine results in this experiment, including average case, the worst case and the best case of the cocktail shaker sort, average case, the worst case and the best case of the insertion sort and the average case, the worst and the best case of the comparison of two algorithms.

There are 51 data points for each experiment.

4.1 Average case of cocktail shaker sort

The result of program is shown in figure 4.1. The graph of average case of cocktail shaker sort is shown in figure 4.2 and 4.3. As shown in picture, the result of time and number of times of basic operations performed are all belong to n^2.

==========Average Case==========

Length	Time	Basic Operation
0	0.0000002	0
1000	0.0023839	758210
2000	0.0094913	3009117
3000	0.0213479	6760438
4000	0.0388531	12076303
5000	0.0626505	18821055
6000	0.0904926	27128777
7000	0.1238739	36662162
8000	0.1639716	48163231
9000	0.2124166	61262272
10000	0.2581118	74995056
11000	0.3134058	90868535
12000	0.3749732	108362349
13000	0.4407106	127050205
14000	0.5135696	147676342
15000	0.5904436	169438071
16000	0.6745838	192716786
17000	0.7638064	216988283
18000	0.8659637	243256822
19000	0.9658768	270612830
20000	1.0780784	301045860
21000	1.1816880	330677658
22000	1.3015951	363456317
23000	1.4250278	398241049
24000	1.5560542	432109426
25000	1.6834079	468218055
26000	1.8257966	507855167
27000	1.9696516	547246046
28000	2.1206304	589473441
29000	2.2741558	632179348
30000	2.4326163	674037451
31000	2.5904187	722281787
32000	2.7645108	772045246
33000	2.9362206	817439531
34000	3.1208049	867683315
35000	3.3168568	921021028
36000	3.4974028	974253953
37000	3.6851803	1026224911
38000	3.9015776	1084579307
39000	4.1055412	1143158904
40000	4.3154130	1202001239
41000	4.5369313	1263692384
42000	4.7691530	1323807564
43000	4.9964299	1390384962
44000	5.2224603	1454536445
45000	5.4661526	1521256786
46000	5.7102758	1589184327
47000	5.9569058	1656146086
48000	6.2232446	1733053290
49000	6.4914388	1803845625
50000	6.7398361	1876121095

Figure 4.1

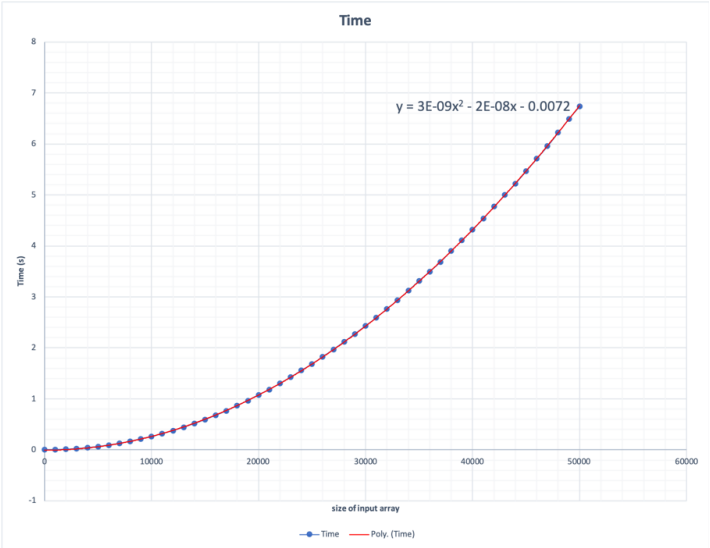

Figure 4.2

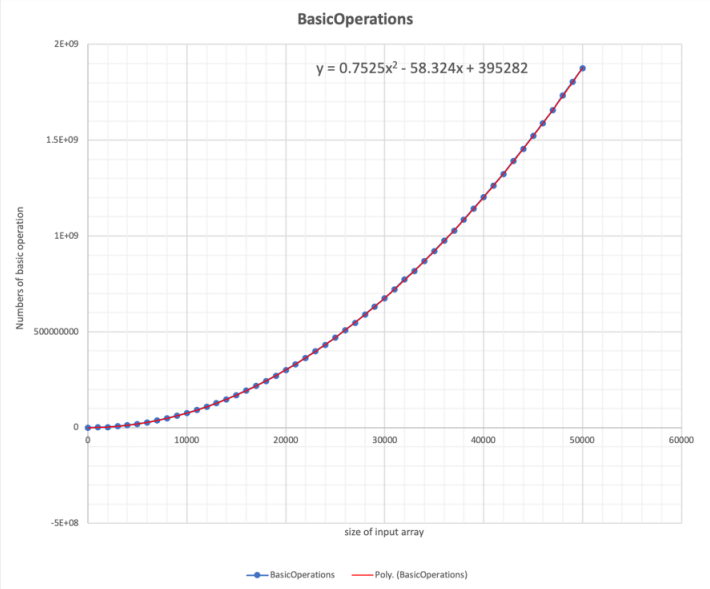

Figure 4.3

4.2 Worst case of cocktail shaker sort

The result of program is shown in figure 4.4. The graph of the worst case of cocktail shaker sort is shown in 4.5 and 4.6. As shown in picture, the result of time and number of times of basic operations performed are all belong to $\Theta(n^2)$. Moreover, the equation of number of basic operations is same as the equation of worst case in section 1.3.4.1.

==========Average Case==========

Length	Time	Basic Operation
0	0.0000002	0
1000	0.0023839	758210
2000	0.0094913	3009117
3000	0.0213479	6760438
4000	0.0388531	12076303
5000	0.0626505	18821055
6000	0.0904926	27128777
7000	0.1238739	36662162
8000	0.1639716	48163231
9000	0.2124166	61262272
10000	0.2581118	74995056
11000	0.3134058	90868535
12000	0.3749732	108362349
13000	0.4407106	127050205
14000	0.5135696	147676342
15000	0.5904436	169438071
16000	0.6745838	192716786
17000	0.7638064	216988283
18000	0.8659637	243256822
19000	0.9658768	270612830
20000	1.0780784	301045860
21000	1.1816880	330677658
22000	1.3015951	363456317
23000	1.4250278	398241049
24000	1.5560542	432109426
25000	1.6834079	468218055
26000	1.8257966	507855167
27000	1.9696516	547246046
28000	2.1206304	589473441
29000	2.2741558	632179348
30000	2.4326163	674037451
31000	2.5904187	722281787
32000	2.7645108	772045246
33000	2.9362206	817439531
34000	3.1208049	867683315
35000	3.3168568	921021028
36000	3.4974028	974253953
37000	3.6851803	1026224911
38000	3.9015776	1084579307
39000	4.1055412	1143158904
40000	4.3154130	1202001239
41000	4.5369313	1263692384
42000	4.7691530	1323807564
43000	4.9964299	1390384962
44000	5.2224603	1454536445
45000	5.4661526	1521256786
46000	5.7102758	1589184327
47000	5.9569058	1656146086
48000	6.2232446	1733053290
49000	6.4914388	1803845625
50000	6.7398361	1876121095

Figure 4.4

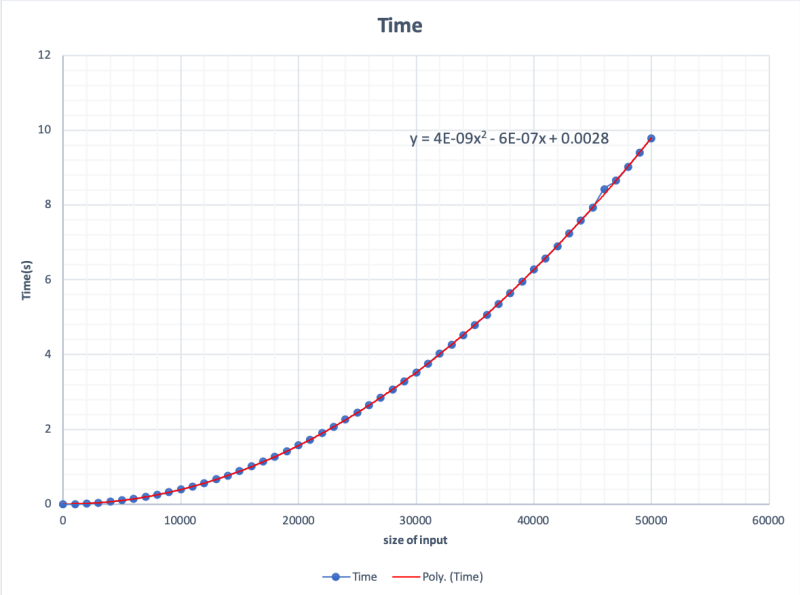

Figure 4.5

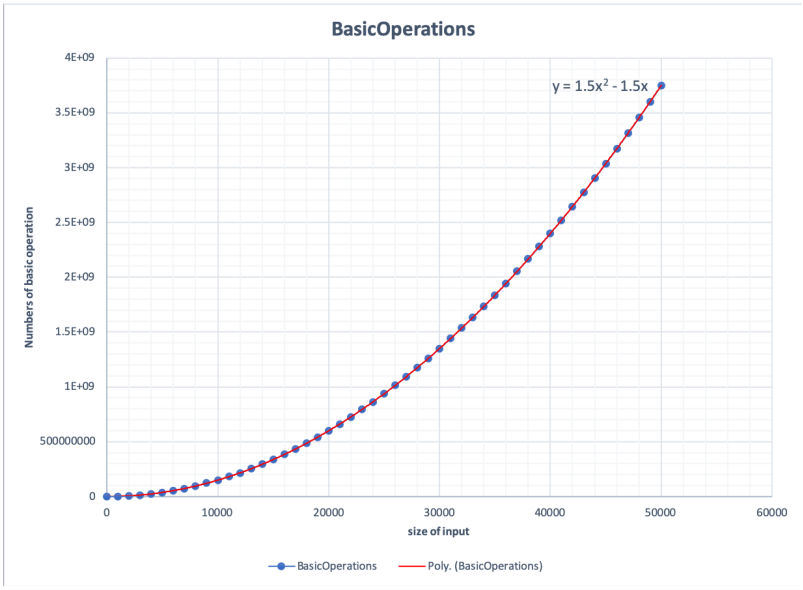

Figure 4.6

4.3 Best case of cocktail shaker sort

The result of program is shown in figure 4.7. The graph of the worst case of cocktail shaker sort is shown in 4.8 and 4.9. As shown in picture, the result of experiment of best case is belong to $\Theta(n)$ which has been discussed in section 1.3.3.2. Moreover, the equation of trendline also similar as the equation which has been described in section 1.3.3.1. In figure 4.5, there are some points stand out from the tread line. It is because the execution time is too short, it might be influenced by other programs. However, it still clear that the time is increased when the length of input array is increased.

```
==================================
=======The Best Case  Start========
==================================
Length     Time       Basic Operation
     0    0.0000003                 0
  1000    0.0000051               999
  2000    0.0000066              1999
  3000    0.0000151              2999
  4000    0.0000115              3999
  5000    0.0000153              4999
  6000    0.0000148              5999
  7000    0.0000195              6999
  8000    0.0000226              7999
  9000    0.0000236              8999
 10000    0.0000269              9999
 11000    0.0000281             10999
 12000    0.0000307             11999
 13000    0.0000302             12999
 14000    0.0000335             13999
 15000    0.0000364             14999
 16000    0.0000416             15999
 17000    0.0000405             16999
 18000    0.0000438             17999
 19000    0.0000469             18999
 20000    0.0000627             19999
 21000    0.0000492             20999
 22000    0.0000502             21999
 23000    0.0000530             22999
 24000    0.0000679             23999
 25000    0.0000611             24999
 26000    0.0000585             25999
 27000    0.0000605             26999
 28000    0.0000639             27999
 29000    0.0000794             28999
 30000    0.0000660             29999
 31000    0.0000684             30999
 32000    0.0000698             31999
 33000    0.0000713             32999
 34000    0.0000756             33999
 35000    0.0000785             34999
 36000    0.0000830             35999
 37000    0.0000840             36999
 38000    0.0000867             37999
 39000    0.0000861             38999
 40000    0.0000904             39999
 41000    0.0000909             40999
 42000    0.0000929             41999
 43000    0.0001084             42999
 44000    0.0001106             43999
 45000    0.0000996             44999
 46000    0.0001008             45999
 47000    0.0001033             46999
 48000    0.0001072             47999
 49000    0.0001290             48999
 50000    0.0001099             49999
```

Figure 4.7

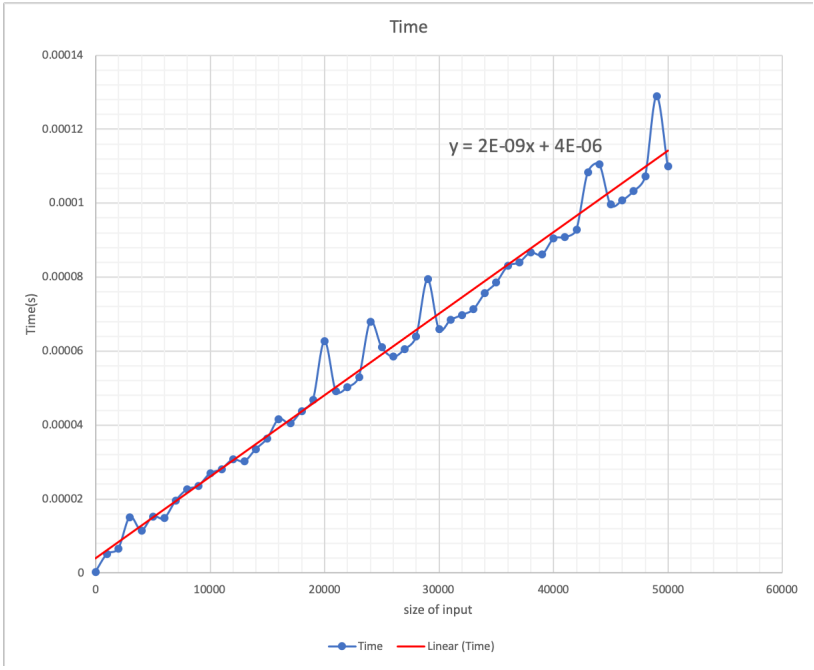

Figure 4.8

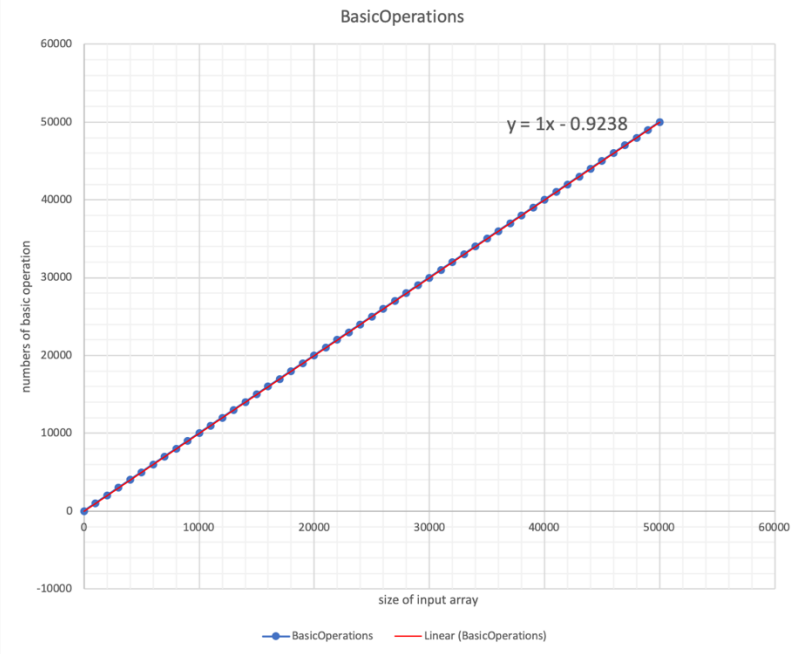

Figure 4.9

4.4 Average case of insertion sort

The result of program is shown in figure 4.10. The graph of average case of insertion sort is shown in figure 4.11 and 4.12. As shown in picture, the result of time and number of times of basic operations performed are all belong to $\Theta(n^2)$.

==========Average Case==========

Length	Time	Basic Operation
0	0.0000021	0
1000	0.0015045	504995
2000	0.0056590	2004040
3000	0.0128503	4511314
4000	0.0222858	8015670
5000	0.0350401	12530216
6000	0.0516070	17997655
7000	0.0679649	24510713
8000	0.0882875	31902056
9000	0.1119808	40547594
10000	0.1381347	49970446
11000	0.1675000	60503831
12000	0.1983166	71905825
13000	0.2344780	84665070
14000	0.2708495	97748430
15000	0.3122720	112435351
16000	0.3535976	127537769
17000	0.3999249	144335715
18000	0.4468526	161933176
19000	0.5007612	180902022
20000	0.5519612	199414559
21000	0.6072256	219819917
22000	0.6693769	241837146
23000	0.7341689	263960358
24000	0.7959416	287570428
25000	0.8683871	313269417
26000	0.9353718	338064849
27000	1.0076913	364770479
28000	1.0840792	392166865
29000	1.1661210	420274075
30000	1.2462536	450341053
31000	1.3338631	480079579
32000	1.4177517	511754188
33000	1.5126051	546289592
34000	1.5996510	576916058
35000	1.6939808	612739580
36000	1.7980359	648395418
37000	1.9014296	685898970
38000	2.0027601	722387431
39000	2.1039864	760403500
40000	2.2172153	799840721
41000	2.3311953	841995151
42000	2.4407089	881342603
43000	2.5588361	922310333
44000	2.6800952	968884833
45000	2.8011678	1010821201
46000	2.9229582	1057406929
47000	3.0595058	1103274862
48000	3.2038563	1153522465
49000	3.3291034	1201544728
50000	3.4622849	1251325884

Figure 4.10

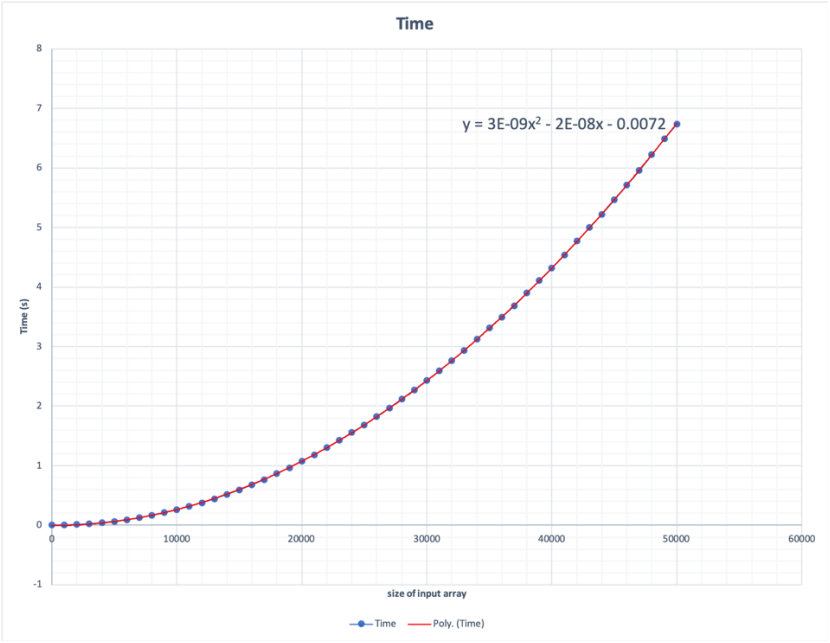

Figure 4.11

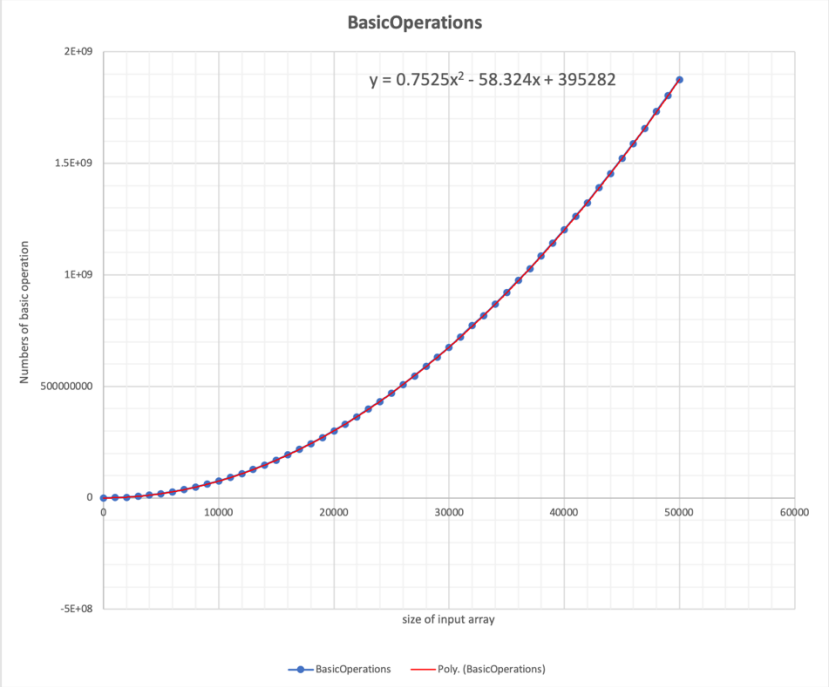

Figure 4.12

6.5 Worst case of insertion sort

The result of program is shown in figure 4.13. The graph of the worst case of insertion sort is shown in 4.14 and 4.15. As shown in picture, the result of time and number of times of basic operations performed are all belong to $\Theta(n^2)$. Moreover, the equation of number of basic operations is same as the equation of worst case in section 2.3.4.1.

```
==================================
=======The Worst Case Start========
==================================
Length     Time       Basic Operation
     0    0.0000025                0
  1000    0.0028716           999000
  2000    0.0111857          3998000
  3000    0.0251914          8997000
  4000    0.0441396         15996000
  5000    0.0739949         24995000
  6000    0.1010259         35994000
  7000    0.1359865         48993000
  8000    0.1765294         63992000
  9000    0.2237697         80991000
 10000    0.2772606         99990000
 11000    0.3347662        120989000
 12000    0.4009660        143988000
 13000    0.4692788        168987000
 14000    0.5433418        195986000
 15000    0.6234705        224985000
 16000    0.7090875        255984000
 17000    0.7991823        288983000
 18000    0.8973552        323982000
 19000    1.0042107        360981000
 20000    1.1076848        399980000
 21000    1.2190603        440979000
 22000    1.3395647        483978000
 23000    1.4618942        528977000
 24000    1.5997386        575976000
 25000    1.7294985        624975000
 26000    1.8731670        675974000
 27000    2.0196187        728973000
 28000    2.1699898        783972000
 29000    2.3342539        840971000
 30000    2.4922757        899970000
 31000    2.6586966        960969000
 32000    2.8344856       1023968000
 33000    3.0106637       1088967000
 34000    3.2068223       1155966000
 35000    3.3873128       1224965000
 36000    3.5940896       1295964000
 37000    3.7916679       1368963000
 38000    3.9978438       1443962000
 39000    4.2246192       1520961000
 40000    4.4272828       1599960000
 41000    4.6576816       1680959000
 42000    4.8849140       1763958000
 43000    5.1160735       1848957000
 44000    5.3577713       1935956000
 45000    5.6259989       2024955000
 46000    5.8666497       2115954000
 47000    6.1144842       2208953000
 48000    6.3801518       2303952000
 49000    6.7045524       2400951000
 50000    6.9242711       2499950000
```

Figure 4.13

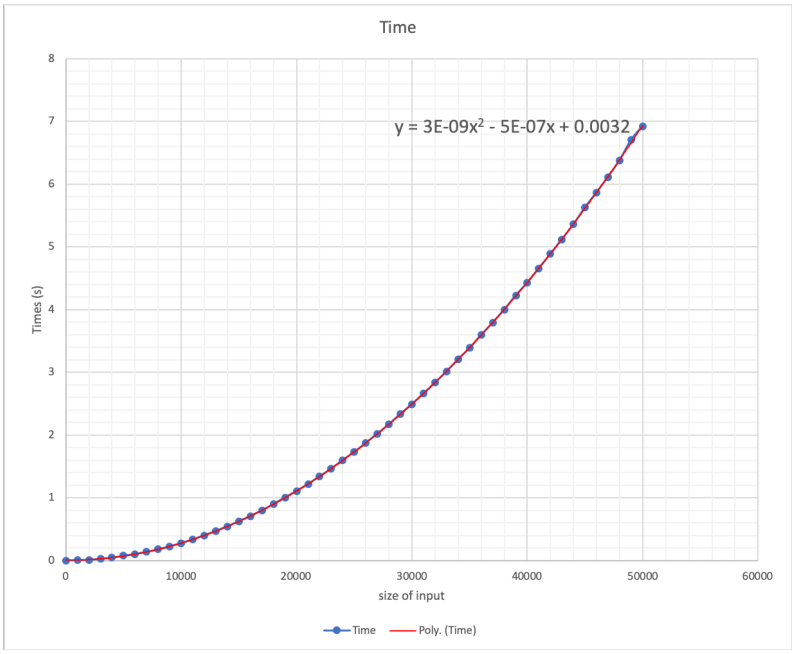

Figure 4.14

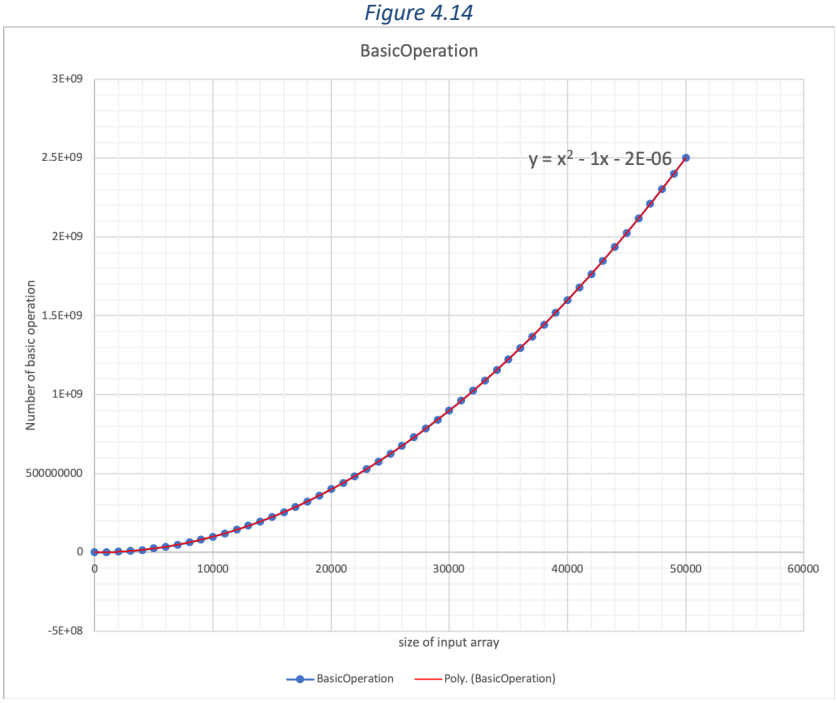

Figure 4.15

4.6 Best case of insertion sort

The result of program is shown in figure 4.16. The graph of the worst case of insertion sort is shown in 4.17 and 4.18. As shown in picture, the number of times of basic operations performed is belong to n. Moreover, the equation of number of basic operations is same as the equation of best case in section 2.3.3.1. In figure 4.17, the trend line of is not clear since the execution time is too short, and it might be influenced by other programs. However, it is still clear that the time is increased when the length of input array is increased.

```
==================================
=======The Best Case  Start========
==================================
Length     Time       Basic Operation
     0    0.0000002                 0
  1000    0.0000028               999
  2000    0.0000069              1999
  3000    0.0000105              2999
  4000    0.0000115              3999
  5000    0.0000242              4999
  6000    0.0000176              5999
  7000    0.0000235              6999
  8000    0.0000257              7999
  9000    0.0000280              8999
 10000    0.0000319              9999
 11000    0.0000299             10999
 12000    0.0000362             11999
 13000    0.0000371             12999
 14000    0.0000559             13999
 15000    0.0000414             14999
 16000    0.0000468             15999
 17000    0.0000511             16999
 18000    0.0000525             17999
 19000    0.0000620             18999
 20000    0.0000607             19999
 21000    0.0000568             20999
 22000    0.0000605             21999
 23000    0.0000600             22999
 24000    0.0000788             23999
 25000    0.0000683             24999
 26000    0.0000718             25999
 27000    0.0000755             26999
 28000    0.0000752             27999
 29000    0.0000780             28999
 30000    0.0000787             29999
 31000    0.0000817             30999
 32000    0.0000842             31999
 33000    0.0000859             32999
 34000    0.0000887             33999
 35000    0.0000967             34999
 36000    0.0001083             35999
 37000    0.0000971             36999
 38000    0.0001143             37999
 39000    0.0001044             38999
 40000    0.0001048             39999
 41000    0.0001045             40999
 42000    0.0001194             41999
 43000    0.0001189             42999
 44000    0.0001142             43999
 45000    0.0001170             44999
 46000    0.0001193             45999
 47000    0.0001274             46999
 48000    0.0001374             47999
 49000    0.0001266             48999
 50000    0.0001407             49999
```

Figure 4.16

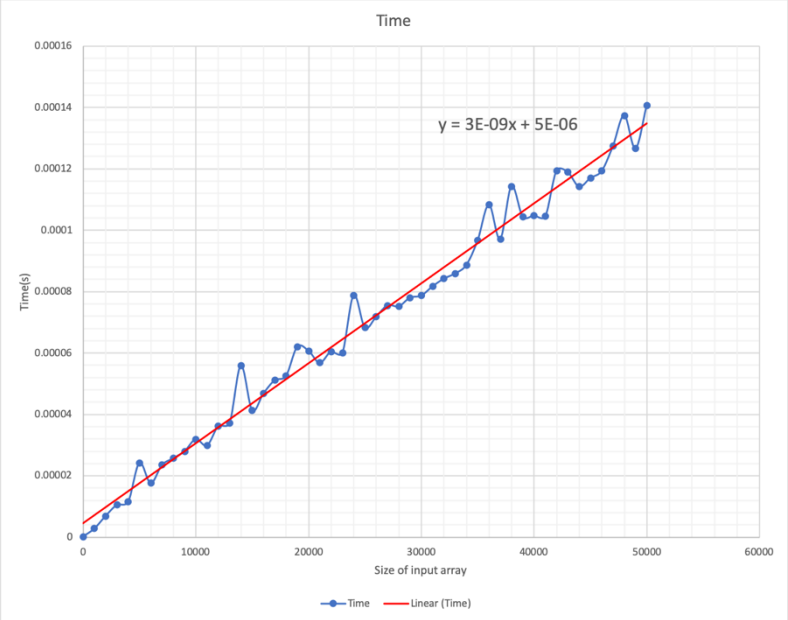

Figure 4.17

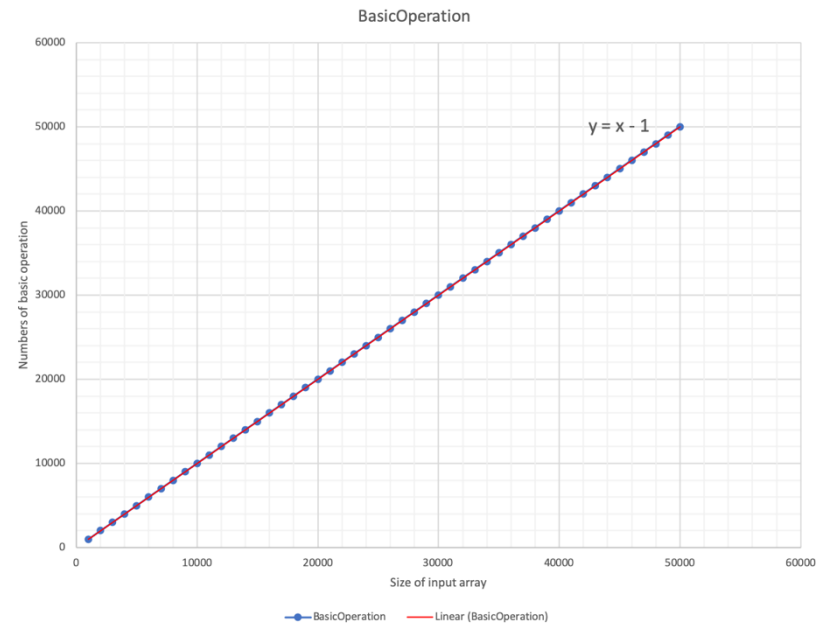

Figure 4.18

4.7 Average case of comparison

The result of program is shown in figure 4.19. The graph of average case of comparison is shown in 4.20 and 4.21. It is clear that in average case, the execution time of cocktail shaker sort is higher than insertion sort.

==========
==========Average==========
==========

Length	Time	Time	Operation	Operation
0	0.0000002	0.0000003	0	0
1000	0.0023198	0.0014014	755836	499080
2000	0.0091353	0.0055485	3019956	2000727
3000	0.0209894	0.0125119	6768121	4490651
4000	0.0396733	0.0227077	12061925	7984679
5000	0.0621484	0.0349923	18824140	12505303
6000	0.0903805	0.0497995	27131857	18066940
7000	0.1250432	0.0674619	36790229	24430671
8000	0.1632300	0.0887318	48178368	32099161
9000	0.2066956	0.1114108	60692430	40327259
10000	0.2600136	0.1385264	75098583	50079169
11000	0.3127748	0.1672909	90881605	60442033
12000	0.3720422	0.1985207	107726754	71688680
13000	0.4400714	0.2336242	127101973	84579987
14000	0.5118576	0.2702095	147052797	97898403
15000	0.5898561	0.3156406	169425641	112780347
16000	0.6747473	0.3551495	192641759	128109183
17000	0.7620961	0.3993324	216191643	144131234
18000	0.8632081	0.4495792	243527812	162267259
19000	0.9610694	0.4990932	271206177	180492795
20000	1.0663674	0.5537950	300588469	200124984
21000	1.1782360	0.6105225	331734647	220738970
22000	1.2971834	0.6737202	364223494	242634262
23000	1.4158967	0.7319090	396647079	264303348
24000	1.5439794	0.7992291	432625484	287782219
25000	1.6763431	0.8643975	469711841	312578749
26000	1.8190772	0.9357612	507819817	337871696
27000	1.9626314	1.0081022	547271660	364085368
28000	2.1064279	1.0844259	588366228	392318061
29000	2.2658819	1.1665897	631374966	420404371
30000	2.4219720	1.2463415	675170415	449458852
31000	2.5808859	1.3278307	720066853	480262061
32000	2.7686072	1.4188239	771425329	512758041
33000	2.9349451	1.5053228	817353440	543971125
34000	3.1187149	1.5995498	866762445	577300489
35000	3.3073216	1.6960837	920332580	612830233
36000	3.4994756	1.7981431	975505209	648595389
37000	3.6947570	1.8947692	1027378910	684618242
38000	3.8928297	1.9991639	1084028098	721510256
39000	4.1086284	2.1126396	1143630622	761479958
40000	4.3222642	2.2193812	1202711267	800950638
41000	4.5278018	2.3249853	1260722151	840608290
42000	4.7564860	2.4467867	1322880986	881031181
43000	4.9887398	2.5604954	1387522098	925098210
44000	5.2436371	2.6861202	1455653969	969508863
45000	5.4672302	2.8052426	1521115842	1011885793
46000	5.7144932	2.9347861	1589698139	1058437578
47000	5.9505140	3.0553356	1654288083	1102537095
48000	6.2214160	3.1876638	1730218306	1151271912
49000	6.4818855	3.3259303	1804909365	1200920084
50000	6.7405552	3.4533734	1875537289	1248084630

Figure 4.19

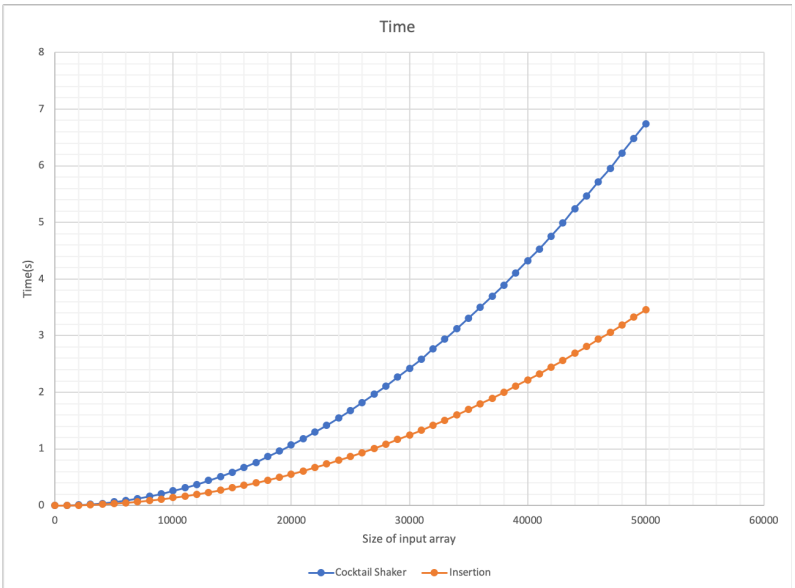

Figure 4.20

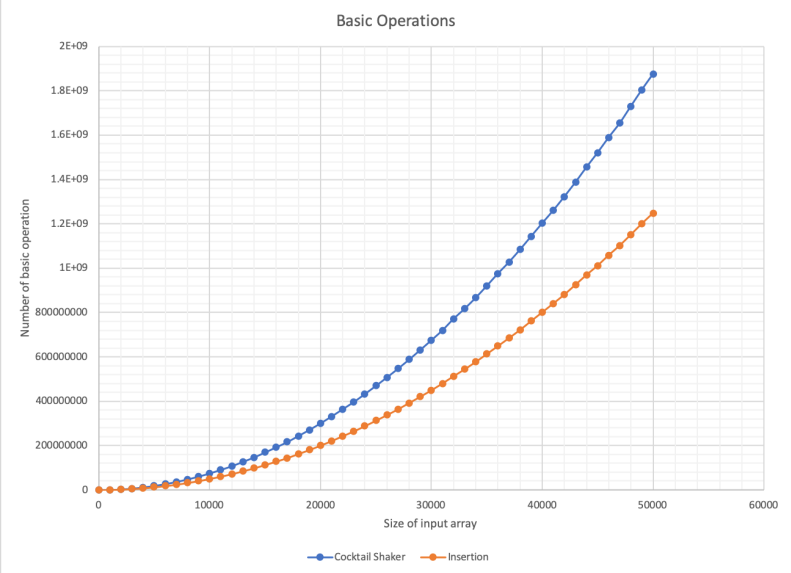

Figure 4.21

4.8 Worst case of comparison

The result of program is shown in figure 4.22. The graph of the worst case of comparison is shown in 4.23 and 4.24. It is clear that in worst case, the execution time of cocktail shaker sort is higher than insertion sort.

```
==================================
===========Worst Case=============
==================================
Length      Time          Time         Operation     Operation
    0    0.0000004     0.0000011             0             0
 1000    0.0040338     0.0028061       1498500        999000
 2000    0.0156699     0.0110867       5997000       3998000
 3000    0.0351273     0.0247653      13495500       8997000
 4000    0.0634388     0.0450914      23994000      15996000
 5000    0.0981084     0.0696037      37492500      24995000
 6000    0.1406918     0.0992989      53991000      35994000
 7000    0.1917111     0.1369608      73489500      48993000
 8000    0.2514764     0.1779082      95988000      63992000
 9000    0.3173738     0.2248813     121486500      80991000
10000    0.3903741     0.2773709     149985000      99990000
11000    0.4739979     0.3343490     181483500     120989000
12000    0.5626166     0.3974133     215982000     143988000
13000    0.6595122     0.4659313     253480500     168987000
14000    0.7654759     0.5418674     293979000     195986000
15000    0.8795977     0.6237492     337477500     224985000
16000    1.0016308     0.7069710     383976000     255984000
17000    1.1296135     0.8008229     433474500     288983000
18000    1.2678541     0.8985541     485973000     323982000
19000    1.4109475     0.9989290     541471500     360981000
20000    1.5650924     1.1064371     599970000     399980000
21000    1.7237766     1.2189945     661468500     440979000
22000    1.8920674     1.3413141     725967000     483978000
23000    2.0708660     1.4685049     793465500     528977000
24000    2.2530795     1.5945242     863964000     575976000
25000    2.4452367     1.7288361     937462500     624975000
26000    2.6462510     1.8744050    1013961000     675974000
27000    2.8543933     2.0232996    1093459500     728973000
28000    3.0679324     2.1680059    1175958000     783972000
29000    3.2949180     2.3255369    1261456500     840971000
30000    3.5267451     2.4922491    1349955000     899970000
31000    3.7620021     2.6600706    1441453500     960969000
32000    4.0083141     2.8325979    1535952000    1023968000
33000    4.2580008     3.0167120    1633450500    1088967000
34000    4.5319669     3.2075002    1733949000    1155966000
35000    4.7887875     3.3893052    1837447500    1224965000
36000    5.0692184     3.5860578    1943946000    1295964000
37000    5.3567134     3.7907530    2053444500    1368963000
38000    5.6496469     4.0016596    2165943000    1443962000
39000    5.9517417     4.2086076    2281441500    1520961000
40000    6.2586655     4.4305066    2399940000    1599960000
41000    6.5880001     4.6575301    2521438500    1680959000
42000    6.8963209     4.8940519    2645937000    1763958000
43000    7.2375009     5.1127921    2773435500    1848957000
44000    7.5798076     5.3634092    2903934000    1935956000
45000    7.9222050     5.6114387    3037432500    2024955000
46000    8.2882476     5.8609391    3173931000    2115954000
47000    8.6352206     6.1207545    3313429500    2208953000
48000    9.0184613     6.3875524    3455928000    2303952000
49000    9.3942208     6.6401819    3601426500    2400951000
50000    9.7880428     6.9207899    3749925000    2499950000
```

Figure 4.22

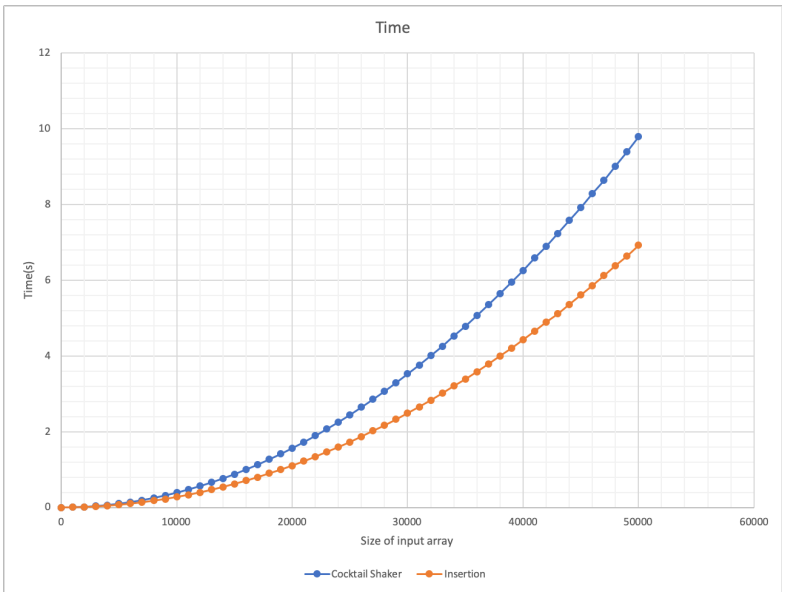

Figure 4.23

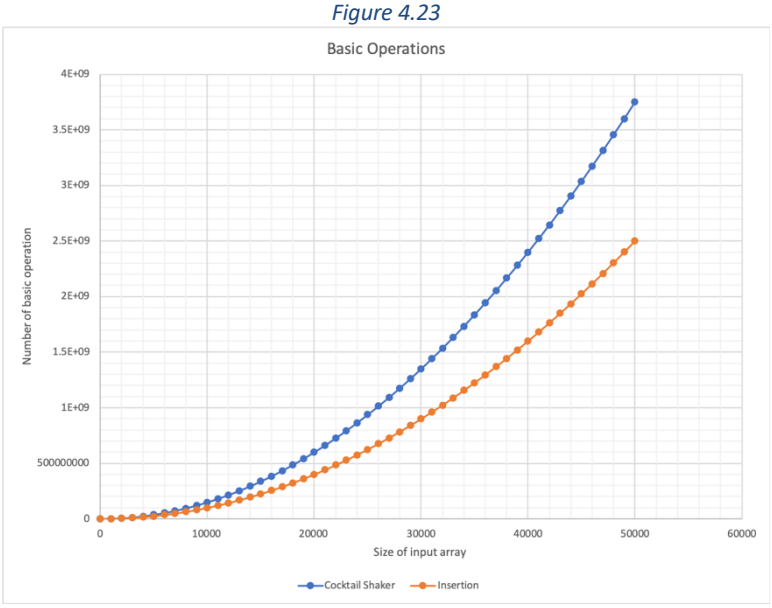

Figure 4.24

4.9 Best case of comparison

The result of program is shown in figure 4.25. The graph of the best case of comparison is shown in 4.26 and 4.27. It is clear that in average case, the number of times of basic operation of cocktail shaker sort is same as insertion sort. In figure 4.27, the result of cocktail sort is under the result of insertion sort, so it is shown in the picture.

```
==================================
============BestCase=============
==================================
Length      Time          Time         Operation     Operation
    0    0.0000007     0.0000006             0             0
 1000    0.0000053     0.0000085           999           999
 2000    0.0000090     0.0000123          1999          1999
 3000    0.0000106     0.0000138          2999          2999
 4000    0.0000160     0.0000150          3999          3999
 5000    0.0000172     0.0000202          4999          4999
 6000    0.0000193     0.0000233          5999          5999
 7000    0.0000220     0.0000265          6999          6999
 8000    0.0000249     0.0000295          7999          7999
 9000    0.0000269     0.0000325          8999          8999
10000    0.0000289     0.0000355          9999          9999
11000    0.0000317     0.0000388         10999         10999
12000    0.0000340     0.0000414         11999         11999
13000    0.0000371     0.0000460         12999         12999
14000    0.0000393     0.0000493         13999         13999
15000    0.0000410     0.0000510         14999         14999
16000    0.0000428     0.0000522         15999         15999
17000    0.0000411     0.0000508         16999         16999
18000    0.0000437     0.0000537         17999         17999
19000    0.0000453     0.0000536         18999         18999
20000    0.0000566     0.0000706         19999         19999
21000    0.0000491     0.0000615         20999         20999
22000    0.0000498     0.0000612         21999         21999
23000    0.0000530     0.0000638         22999         22999
24000    0.0000551     0.0000679         23999         23999
25000    0.0000572     0.0000691         24999         24999
26000    0.0000624     0.0000798         25999         25999
27000    0.0000634     0.0000740         26999         26999
28000    0.0000641     0.0000826         27999         27999
29000    0.0000654     0.0000826         28999         28999
30000    0.0000690     0.0000848         29999         29999
31000    0.0000754     0.0000992         30999         30999
32000    0.0000715     0.0000900         31999         31999
33000    0.0000731     0.0000929         32999         32999
34000    0.0000817     0.0000981         33999         33999
35000    0.0000808     0.0000982         34999         34999
36000    0.0000843     0.0001013         35999         35999
37000    0.0000806     0.0001033         36999         36999
38000    0.0000972     0.0001200         37999         37999
39000    0.0000937     0.0001077         38999         38999
40000    0.0000894     0.0001108         39999         39999
41000    0.0000910     0.0001201         40999         40999
42000    0.0000958     0.0001175         41999         41999
43000    0.0000961     0.0001248         42999         42999
44000    0.0000946     0.0001217         43999         43999
45000    0.0001020     0.0001273         44999         44999
46000    0.0001038     0.0001319         45999         45999
47000    0.0001039     0.0001263         46999         46999
48000    0.0001047     0.0001349         47999         47999
49000    0.0001116     0.0001352         48999         48999
50000    0.0001105     0.0001317         49999         49999
```

Figure 4.25

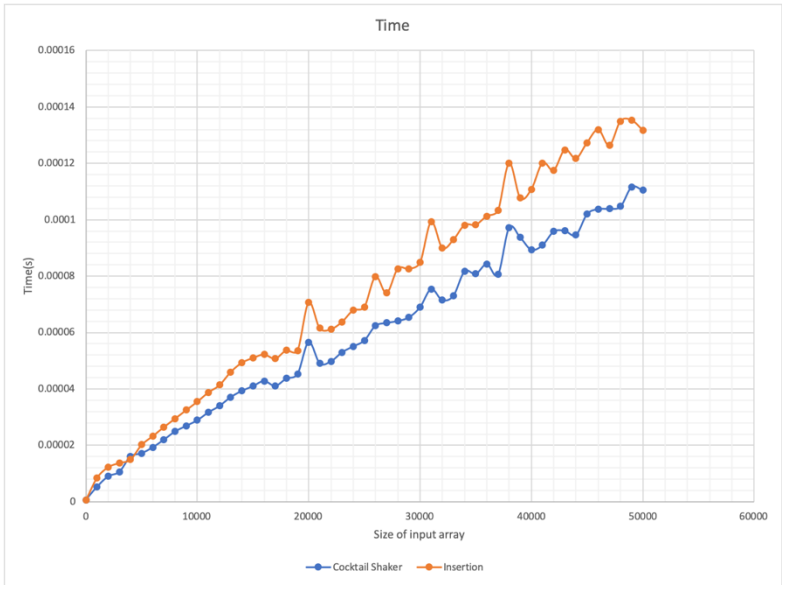

Figure 4.26

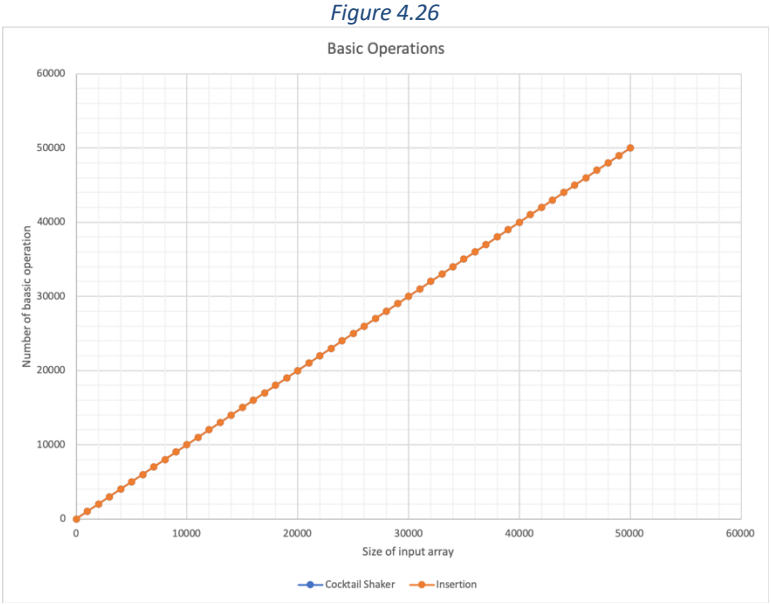

Figure 4.27

5. Conclusion

The results of the worst case and average case of cocktail shaker sort and insertion sort match the efficiency class $\Theta(n^2)$ which has been discussed in section 1 and 2. Moreover, the results of the best case of cocktail shaker and insertion sort also match the efficiency class $\Theta(\mathrm{n})$ which has been discussed in section 1 and 2. However, the graph of execution time of the best case is not exact linear. There are some data points are stand out of the line. The reason is that when the program executed, there are also other programs is executed in the computer. Moreover, the time of execution time is too short, so the impact is apparent. However, the trendline is still linear. The number of times of basic operation performed in the worst case and the best case for each sort algorithm is similar as the equation which has been described in section 1 and section 2.

The result of comparison clearly shows that cocktail shaker sort needs more time and performed more times in average case and worst case. In the worst case, as we have discussed in the section 1 and 2, although both two algorithms are $\Theta(n^2)$, cocktail shaker sort still needs to perform more times of basic operation than insertion sort. This situation can overserve obviously in the graph. In the best case, as we have discussed in the section 1 and 2, the number of times of basic operation performed in two algorithms are the same. This situation is shown clearly in the figure 4.25 and 4.27. In the figure 4.27 the blue line which represents the cocktail shaker sort is totally covered by the orange line which represents the insertion sort. However, the time of insertion sort is higher than cocktail shaker sort in figure 4.26. This is because there are also other instructions need to be performed in each algorithm. For example, in the insertion sort, the “j = i” performed n-1 times but in cocktail shaker sort, the “swapped = false” just performed once. Although these operations are not basic operation, they still cause great impact when the execution time is short. As a result, the execution time of the best case of insertion sort is higher than cocktail shaker sort since the insertion sort executed more other operations.